COMSC-210 Lecture Topic 16
Graphs

□ Reference

Childs Ch. 15
YouTube DFS

□ Graphs

Generalization of linked list concept
nodes can link to *any* other node
...including self
applications:
route maps
project charts
mazes

□ Graph Terminology

"vertex" (like a linked list node)
"edge" (like a linked list link)
"cost" of taking an "edge" btwn 2 "vertices"
1 or distance or time or...
"digraph" -- directed links (one-way links)
"graph" -- bi-directional links
"search" -- iteration
"route"

□ Representing Vertices

```
struct Vertex
{
  string name; // not a template
  other stuff
};
```

...more data members will be added later...

□ Storing Vertices And Edges

use any O(1) index-accessible array (incl. STL representations)
store vertices in an indexed array structure
e.g., main "database" as vector
`vector<Vertex> database;`
use push_back to populate the array
e.g., main "database" as array
`Vertex database[N];`

for *dense* network (many vertex interconnections)
store edges in an "adjacency matrix": 2D table
row for each "from" vertex
column for each "to" vertex
cell contains "cost", if connected
triangular or square
e.g., `vector<vector<double> > cost;`
or `double cost[N][N];`

for *sparse* network (vertices connect to neighbors mostly)
store edges in "adjacency lists": lists of downstream vertices
stored with vertices, in struct definition
`list<pair<int, double> > adjacentVertices;`
STL pair includes index of "neighbor" vertex...
...and weighted edge cost to that neighbor
Req'd space between `>` *symbols*

□ Implementation Of Adjacency Lists For Sparse Networks

```
struct Vertex
{
  string name;
  list<pair<int, double> > adjacentVertices;
  other stuff
};
```

□ Graph Iteration Techniques

simplest: iterate "database" list
the familiar O(n) linear search
more complicated: BFS and DFS
from a start vertex through all *reachable*
choose *any* starting point
breadth first ("fans out" from start)
depth first ("probes" from start)
need to avoid revisiting vertices
all unit costs -- no cost list needed:

□ Finding Routes

from a start vertex to an ending vertex
goal: function to *return cost of route AND STL stack with vertex names*
need to avoid revisiting vertices & track how I got there:

```
struct Vertex
{
  string name; // constant
  list<pair<int, double> > adjacentVertices;

  bool isVisited; // variable
  int prev; // variable (sentinel = -1)
  double cost; // variable
};
```

problem: how to return 2 things?
an STL solution: `pair<stack<int>, double>` as return type (requires "utility" library)

```
pair<stack<int>, double> getRoute(int iStart, int iEnd, vector<Vertex> database
{
  pair<stack<int>, double> result; // route, cost
  ...
  result.second = database[iEnd].cost;
  for (int vertex = iEnd; vertex >= 0; vertex = database[vertex].prev)
    result.first.push(vertex);
  return result;
}
```

□ Finding The "Shortest Route"

"cost" is 1 per edge -- edges are *not* weighted
that is, count the number of "jumps"

```
set cost=0 and previous index=-1 for all vertices, and mark all as not visited
create a queue to store the "to do list"
mark start vertex as "visited" and push it onto the "to do list"
while the "to do list" is not empty
 peek/pop a vertex from the "to do list"
 for each of that vertex's neighbors
  if neighbor has not yet been visited
   mark neighbor as visited
   set neighbor's cost to 1+cost of vertex under consideration
   set neighbor's previous vertex to the vertex under consideration
   push neighbor into the "to do list"
  if neighbor vertex contains the index of the end city
   empty the "to do list"
   exit for-loop
the route's cost equals the end vertex's cost
build a stack of entries, starting from the end vertex, back towards the start
```

□ Finding The "Cheapest Route" -- Dijkstra's Algorithm

edge "costs" are weighted
revisited vertex *may* be cheaper -- need to track

```
struct Container // vertex container, for multiple ways to get to a vertex
{
  int vertex; // index in database array
  int prev; // index in database array
  double cost;

  bool operator<(const Container& v) const
  {
    return cost > v.cost;
  }
};
```

need operator< for use in priority queue

```
reset the cost and previous indices for all vertices, isVisited to false
create priority queue of Container objects
create a Container object for the start vertex, with 0 cost and negative prev index
push the start vertex's container onto the priority queue
while the priority queue is not empty
 peek/pop a container object from the priority queue
 if contained object's been visited, "continue"
 mark contained object as visited
 set its cost to that of its container
 set its prev to that of its container
 if contained vertex is the end vertex, exit while-loop
 for each of that vertex's unvisited neighbors
  create a container object for neighbor
  set its cost to that of the vertex, plus edge cost
```

```
struct Vertex
{
  string name; // constant
  list<pair<int, double> > adjacentVertices;

  bool isVisited; // variable flag used in searches
};
```

□ Breadth First Search (BFS)

goal: function to *return STL queue of all vertices reachable from the start*

```
create an empty result queue to return at end of function call
create another queue to store the "to do list"
initialize each vertex in database: set to "not visited"
mark starting vertex as "visited"
push start vertex's index onto the result queue
push start vertex's index onto the "to do list"
while the "to do list" is not empty
  peek/pop a vertex from the "to do list"
  for each of that vertex's "neighbors"
    if neighbor has not yet been visited
      mark neighbor as visited
      push neighbor onto the result queue
      push neighbor onto the "to do list"
```

□ Depth First Search (DFS)

goal: function to *return STL queue of all vertices reachable from the start*

```
create an empty result queue for returning at the end
create a stack to store the "to do list"
initialize each vertex in database: set to "not visited"
push start vertex's index onto the "to do list"
while the "to do list" is not empty
  peek/pop a vertex from the "to do list"
  if that vertex has not yet been visited
    mark the vertex as visited
    push vertex onto the result queue
    for each of that vertex's "neighbors" (in reverse order)
      if neighbor has not yet been visited
        push neighbor onto the "to do list"
```

```
  set its previous to the vertex
  push container onto priority queue
the route's cost equals the end vertex's cost
build a stack of entries, starting from the end vertex, back towards the start
```

why a container? O(n) to find vertex in queue, pop, change, and reinsert

why not quit when end vertex found? might still be shorter routes

□ The Main Program

```
// load database
vector<Vertex> database;
...

// get start/end points
string startCity, endCity;
int iStartCity = iEndCity = -1; // indices of start/end cities
cout << "\nEnter the source city [blank to exit]: ";
getline(cin, startCity);
if (startCity.length() == 0) break;

cout << "Enter the destination city [blank to exit]: ";
getline(cin, endCity);
if (endCity.length() == 0) break;

for (int i = 0; i < database.size(); i++)
{
  if (startCity == database[i].name) iStartCity = i;
  if (endCity == database[i].name) iEndCity = i;
}

// get cheapest route
if (iStartCity >= 0 && iEndCity >= 0)
{
  pair<stack<int>, double> result = getCheapestRoute(iStartCity, iEndCity, da
  cout << "Total miles: " << result.first;
  while (!result.second.empty())
  {
    cout << " " <<
    database[result.second.top()].name << ' ';
    result.second.pop();}
} }
cout << endl;
```

...or use `database[iEndCity].cost` as route's cost and avoid using `pair`